Contents

Chapter 1

GNU Radio

1.1 Introdution

GNU Radio[1] is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in hobbyist, academic and commercial environments to support both wireless communications research and real-world radio systems.

A software radio is a radio system which performs the required signal processing in software instead of using dedicated integrated circuits in hardware. The benefit is that since software can be easily replaced in the radio system, the same hardware can be used to create many kinds of radios for many different transmission standards; thus, one software radio can used for a variety of applications.

GNU Radio performs all the signal processing. You can use it to write applications to receive data out of digital streams or to push data into digital streams, which is then transmitted using hardware. GNU Radio has filters, channel codes, synchronization elements, equalizers, demodulators, vocoders, decoders, and many other elements (in the GNU Radio jargon, we call these elements blocks) which are typically found in radio systems. More importantly, it includes a method of connecting these blocks and then manages how data is passed from one block to another. Extending GNU Radio is also quite easy; if you find a specific block that is missing, you can quickly create and add it.

Since GNU Radio is software, it can only handle digital data. Usually, complex baseband samples are the input data type for receivers and the output data type for transmitters. Analog hardware is then used to shift the signal to the desired center frequency. That requirement aside, any data type can be passed from one block to another - be it bits, bytes, vectors, bursts or more complex data types.

GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions, where available. GNU Radio Companion(GRC) is a Simulink-like graphical tool to design signal processing flow graphs.

GNU Radio supports several radio front-ends, either natively or through additional out-of-tree modules. We will be using Ettus Research USRP platform and RTL-SDR TV tuners.

1.2 First flow-graph

Flow graphs are graphs (as in graph theory) through which data flows. Many GNU Radio applications contain nothing other than a flow graph. The nodes of such a graph are called blocks, and the data flows along the edges.

Any actual signal processing is done in the blocks. Ideally, every block does exactly one job - this way GNU Radio stays modular and flexible. Blocks are usually written in C++ (might also be Python); writing new blocks is not very difficult.

In order to illuminate this diffuse topic a little, let's start with an example. Our aim is to generate a sinusoid and play it on audio output. Open GNU Radio companion by executing gnuradio-companion on a terminal. The completed flow-graph is shown in Figure 1.1.

The blocks are connected at ports. The first block has no input port, it produces samples. Such a block, with only output ports, is called a source. In an analog fashion, the final block, with no outputs,

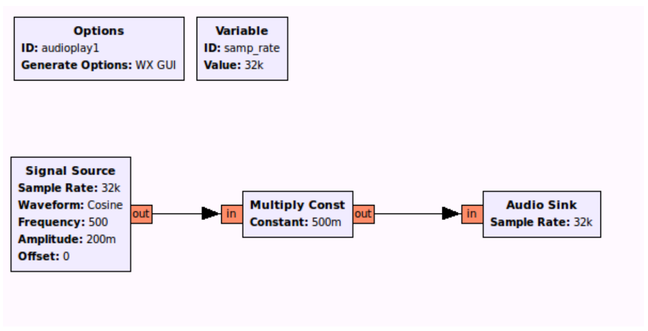


Figure 1.1: Playing 500Hz sinusoid on audio output. Take signal source and audio sink block from the list of blocks. Set the data type of signal source to float. Set the frequency to 500Hz and sampling rate to 32k. Set the sampling rate in the audio sink block to 32k. Connect these blocks and execute the flow-graph.

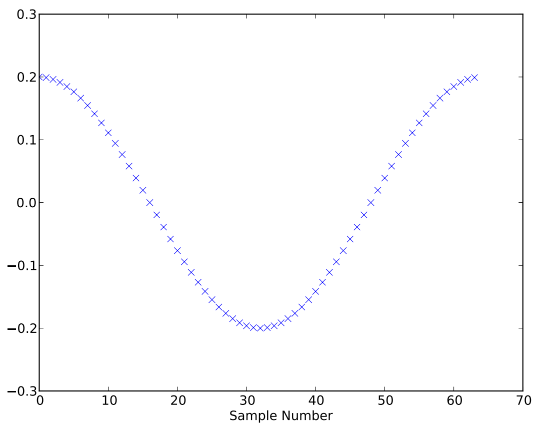


Figure 1.2: The first 64 samples of output from the signal source.

is called a sink.

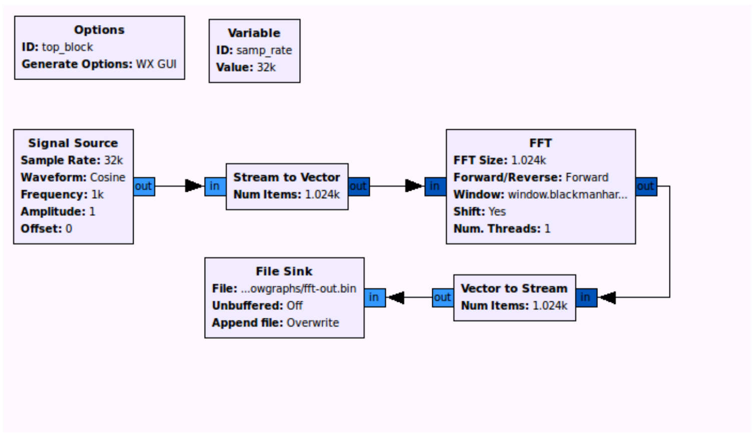
So what's happening here? The signal source produces samples of 500Hz cosine sampled at 32kHz per second. The first 64 samples (which corresponds to a complete cycle of cosine in this case) is plotted in Figure 1.2. These samples are connected to the multiply constant block, which multiplies each of the input with 0.5. Finally, the samples are passed to the audio output block which plays the incoming samples using computers audio card.

In general, we call whatever a block outputs an item. In Figure 1.1, one item was a float value representing one sample produced by the signal source. However, an item can be anything that can be represented digitally. The most common types of samples are real samples (as before), complex samples (the most common type in software defined radio), integer types, and vectors of these scalar types.

In Figure 1.1, apart from playing the samples on audio card, everything else is done in software. Since the audio card can only take specific sampling rates(32kHz, 44.1kHz, etc) the rate at which the samples are processed through this flow graph will be equal to the sampling rate of the audio card.

Let us consider another example of computing the FFT of a signal source. The flow graph is given in Figure 1.3. In this example, the second block (stream to vector) produces one item(which is a vector of 1024 complex number) for every 1024 input items. So, the rate at which it produces items is 1024 times smaller then the rate at which it consumes items (the fact that it actually produces bytes at the same rate it consumes them is irrelevant here). Such a block is called a decimator, well, because it decimates the item rate. A block which outputs more items than it receives is called an interpolator. If it produces and consumes at the same rate, it's a sync block.

Let us ask the question what is the base sampling rate in Figure 1.3? Or how many complex numbers produced by the signal source, will be processed through this flow graph in a second? Well, we can't really answer this question. In this flow graph all that we are do, is to generate some numbers, do some math on the generated numbers, and save the output to a file. As long as there is no hardware clock



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 32k

Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0

Stream to Vector
Num Items: 1.024k

FFT
FFT Size: 1.024k
Forward/Reverse: Forward
Window: window.blackmanhar...
Shift: Yes
Num. Threads: 1

File Sink
File: ...owgraphs/fft-out.bin
Unbuffered: Off
Append file: Overwrite

Vector to Stream
Num Items: 1.024k

Figure 1.3: Computing FFT

present which fixes the rate, sampling rate is meaningless–only relative rates (i.e. input to output rates are important. The computer may handle the samples as fast as it wants. Note that this can cause the computer to lock up by allocating 100% of CPU cycles to your signal processing.

In order to avoid the computer getting busy doing all the math, it is recommended to use a throttle block when we run GNU Radio without any other hardware. The throttle block is tied to computer's hardware clock, and it throttles the number of items passing through it.

Also when we have hardware involving different sampling rates, we might have to use some up-sampling/down-sampling so that we meet the sampling rate requirement of all the hardwares.

Appendices

```python
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Audioplay1
# Generated: Sat Jun 28 18:50:14 2014
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class audioplay1(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Audioplay1")
        _icon_path = "/usr/share/icons/hicolor/32x32/apps/gnuradio-grc.png"
        self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vff((0.5, ))
        self.audio_sink_0 = audio.sink(samp_rate, "", True)
        self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE,
    500, 0.2, 0)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.audio_sink_0, 0))
        self.connect((self.analog_sig_source_x_0, 0), (self.blocks_multiply_const_vxx_0,
 0))



    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = audioplay1()
    tb.Start(True)
    tb.Wait()
```

Listing 1: The python program corresponding to the flow-graph in Figure 1.1

Bibliography

[1] http://gnuradio.org

[2] http://www.linear.com/product/LTC5598

[3] http://www.linear.com/product/LTC6946